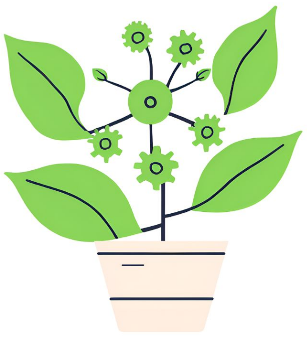
Smart Plants

Gabriele Romano, Giulio Saulle
Professor: Prof. William Fornaciari

AY 2023/2024



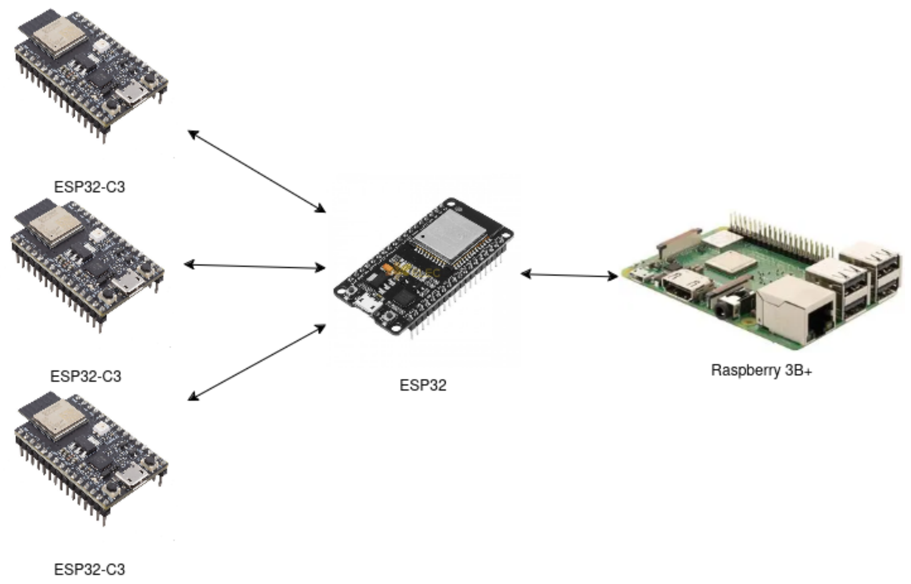
POLITECNICO
MILANO 1863

Contents

1 Overview

This project was created to help people in managing their balcony/vegetable garden, being able to have a smart dashboard in which plant parameters such as temperature, humidity, soil moisture, and brightness will be showed in a user-friendly interface. The primary goal of this system is to streamline and automate plant care, providing users with an effortless way to monitor and manage their plants' health. This intelligent approach not only enhances plant care but also reduces the time and effort needed for maintenance, allowing users to enjoy healthier plants with minimal involvement.

2 System Architecture



The system consists of three main components:

- **Plant Sensor (ESP32-C3)**: Monitors the plant's soil moisture, temperature, humidity, and light conditions through sensors. May activate electro valve to water the plant.
- **Plant Brain (ESP32)**: Acts as a central hub that receives data from the plant sensor, computes message to display in smart mirror and send it, moreover it sends room temperature and humidity in the same MQTT topic.
- **Smart Mirror (Raspberry Pi 3B+)**: Displays data on a dashboard, providing a visual representation of plant conditions and room parameters. The software is based on Raspian, but test run has been conducted on Ubuntu 24.0 and shows no decreases in terms of performance or usability.

3 Hardware Setup

The following hardware components are used:

- **Raspberry Pi 3 B+**: The central processing unit responsible for managing the Smart Mirror display, a program which runs in a Debian/Raspian operative systems. The chip-set is used to manage the display part on remote, so there is no need for a portable device, thus it can be connected to the home power grid.
- **Screen device**: Displays the dashboard with plant health metrics (this is specified since the task is not only feasible by computers, thus a screen module connected to the Raspberry or another board, given the possibility of customizing the UI, is always possible).
- **ESP32-C3 (Plant Sensor)**: Responsible for managing collected data from the sensors attached to the plant. Relays the data to brain sensor.
- **Blood Electrovalve**: a small electrovalve alimented with 3V, used to irrigate the plants form remote.
- **ESP32 (Plant Brain)**: Manages sensor data and communication with the Smart Mirror.
- **Sensors**:
  - **Soil Moisture Sensor(Capacitive Moisture Soil Sensor v1.2)**: Measures soil moisture levels.
  - **Humidity and Temperature Sensor(DHT11)**: Tracks ambient temperature and humidity.
  - **Light Sensor(TSL2561)**: Monitors light levels around the plant.
- **Power Supplies/Transformers**: Esp32 is supplied by a current power adapter, like the raspberry PI3B+, while esp32-C3 is supplied by a 2032 battery model, through a correct socket.

4 Schematics

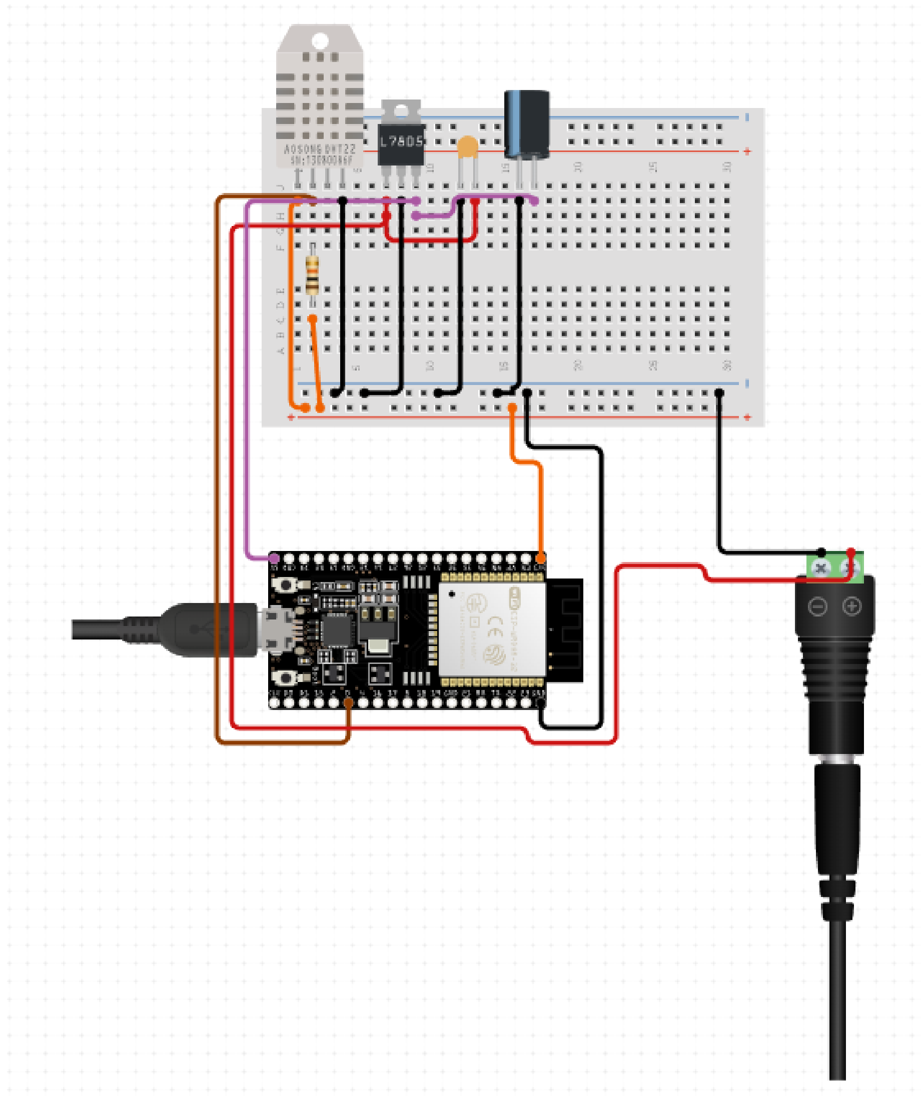


Figure 1: ESP brain

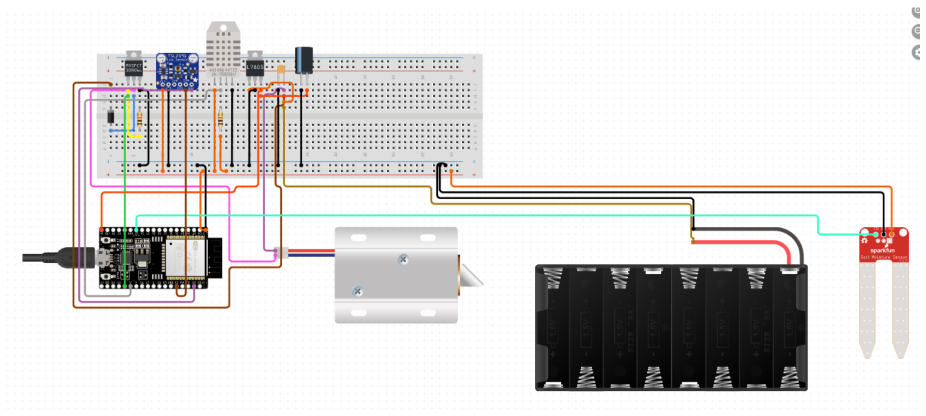


Figure 2: ESP sensor

5 Software components

The software is designed to enable efficient data acquisition and communication between components.

- **Raspbian OS**: Runs on the Raspberry Pi, providing the program for the Smart Mirror.
- **Smart Mirror**: or to address what we vbased our project, Magic Mirror. It's an open source modular smart mirror platform, using Electron as a wrapper for the numerous modules.
- **C for Sensors**: Low-level language for handling sensor data collection and sending it via MQTT.
- **Smart Mirror Dependencies**:
  - **default modules**: As clock and weather, are the ones that could be useful to our task. In particular these to have been modified to show the temperature and time of Italy (in particular Milan).
  - **MMM-MQTTbridge**: Handles MQTT communication for data synchronization between the plant brain and Smart Mirror. The way to do this is to utilize a infra communication "network" of the Smart Mirror: Notifications! Then we assist at an exchange of messages through the broker of messages from our plants.
  - **custom modules**: having the possibility to implement our personal methods permitted us to show at display exactly what we wanted about plants and rooms info. These modules are very linear and ultimately consists in two interfaces given by two files respectively, a and a CSS.

6 Communication

In order to enable seamless, real-time communication between the different components of the Smart Plants system, MQTT (Message Queuing Telemetry Transport) is used as the primary communication protocol. MQTT is specifically chosen for its lightweight nature and low power consumption, which is ideal for resource-constrained devices like the ESP32. It allows for efficient, reliable messaging with minimal overhead, making it particularly suitable for applications like this, where data packets from sensors need to be sent frequently without draining power or using excessive bandwidth.

The MQTT protocol operates in a publish-subscribe model, which provides the flexibility needed for this system's architecture. Multiple devices can subscribe to relevant channels, receiving only the data they need. This helps in minimizing data traffic while ensuring timely updates. The use of MQTT ensures that even in low-bandwidth or unreliable network conditions, messages will be transmitted efficiently and securely between devices. In particular we relay on the broker "**mqtt-dashboard.com**" (while for debugging we used "**hivemq**") that offers a stable platform to relay data through a public channel (given there are no sensible data), this allows us to have a long distance communication, means the user can control his balcony plants and his far away garden at the same time!

There are three main MQTT channels in the system:

- **Smart_plants**: This channel is dedicated to transmitting critical sensor data such as temperature, humidity, light, and soil moisture from the Plant Sensor to the Plant Brain. By consolidating all environmental data into a single stream, the system simplifies processing while maintaining accurate and up-to-date readings.
- **Smart_plants_debug**: This channel is reserved for sending error messages and debugging information, allowing for the identification and resolution of issues in the sensor network. By isolating debug information in a separate channel, the system remains robust without cluttering operational data streams.
- **Smart_mirror**: This channel is used to transmit notifications and plant data from the Plant Brain to the Smart Mirror. The Smart Mirror subscribes to this channel to receive updates, which it then displays in real-time, providing the user with a clear and accessible overview of their plants' conditions.

By leveraging MQTT, the system maintains efficient communication, reduces latency, and ensures that critical updates are reliably transmitted, even in network-challenged environments.

7 Data Flow

7.1 Plant Sensor (Data Collection)

The Plant Sensor (ESP32-C3) continuously monitors soil moisture levels at intervals of M seconds. When the soil becomes too dry, the system automatically activates a pump to irrigate the plant until either the moisture level returns to an acceptable range or the maximum watering duration (MAX_WATERING_TIME) is reached.
Conversely, if the soil becomes excessively wet, the watering_time is set to -1, signaling the plant's control system to alert the user to relocate the plant to a drier environment.

In addition to soil moisture, the Plant Sensor collects data on light, temperature, and humidity every N seconds, categorizing these readings into three status levels:

- **0** = too low.
- **1** = optimal.
- **2** = too high.

Plant sensor messages are formatted as **JSON**:

```
{
  "room": n, //value which addresses in which "room" sensors are located
  "plant": "plant_name", //which plant the message is referring to
  "plant_img":"plant_image", //a link to a plant image useful for smart
      mirror
  "sensors": {
    "soil_moisture": soil_moisture_value,
    "temperature": temperature_value,
    "humidity": humidity_value,
    "light": light_value
  }, //values detected from sensors
  "watering_time": "watering_time", //can be -1 if soil is too wet,
      otherwise watering time in seconds is indicated
  "status": {
    "temperature": temp_status,
    "humidity": hum_status,
    "light": light_status
  } //values computed from values detected and optimal data
}
```

7.2 Plant Brain (Data Processing and Forwarding):

In addition to environmental monitoring, the Plant Brain is responsible for generating comprehensive status updates regarding the plant's well-being. Current plant parameters are retrieved from Plant Sensor message, then is packaged into a user friendly message, which is then sent to the Smart Mirror for visibility. This allows users to always be informed about the plant's condition. The format of these JSON messages is structured as follows:

```
{
  "plant": "plant_name", //name of the plant
  "plant_img": "plant_img", //image url of the plant we want to show
  "watering_time": "watering_time", //value referring to the amount of
      time the plant was watered for
  "message": "message" //user friendly message
}
```

Below are some typical examples of the messages sent to the Smart Mirror, providing real-time feedback on both environmental conditions and the plant's status:

```
"Brrr... it's cold here, but it's bright. Can we find a warmer
    place?",

"Yay! Everything feels just right! I'm feeling good.",

"Phew... it's too hot and bright here, but the humidity is okay. Can
    you find a cooler place?",
```

By integrating environmental monitoring with proactive messaging, the Plant Brain ensures optimal care for the plant while keeping the user informed in a timely manner through the Smart Mirror.

7.3 Smart Mirror (Data Display):

The Smart Mirror receives the messages from the Plant Brain and displays them in a dashboard-like interface. The system visualizes plant health, including watering status and sensor readings, offering an intuitive understanding of plant conditions in real time. MMM-MQTT module take in messages by connecting to the broker mqttServer: *"mqtt://mqtt-dashboard.com:1883"* on topic *smart_mirror*, then it handles the payload by publishing it to the internal topic "PLANT_INFO" for plant information and "ROOM_INFO" for room information:

```
var mqttHook = [
    {
      mqttTopic: "smart_mirror",
      mqttPayload: [
        {
          mqttNotiCmd: ["Command 1"]
        },
      ],
    },
  ];
var mqttNotiCommands = [
    {
      commandId: "Command 1",
      notiID: "PLANT_INFO",
    },
    {
      commandId: "Command 1",
      notiID: "ROOM_INFO",
    },
  ];

module.exports = { mqttHook, mqttNotiCommands};
```

Then we have the custom modules that parse the message arrived as notification, still in JSON format, check if the fields of the message are correctly received and then build the list to display with few adjustments to do it correctly (Plant Info is shown as an example given that Room Info has a similar handling of messages, but different topic):

```
[...]
  notificationReceived: function (notification, payload, sender) {
    if (notification === "PLANT_INFO") {
      console.log("Ricevuto PLANT_INFO:", payload);

      if (typeof payload === "string") {
        try {
          payload = JSON.parse(payload);
        } catch (e) {
          console.error("Errore nel parsing del JSON:", e);
          return;
        }
      }

      this.updatePlantData(payload);
    }
  },

  updatePlantData: function (plantData) {
    console.log("Aggiornamento dati pianta:", plantData);
    const plantListContainer =
        document.querySelector(".plant-list-container");

    if (!plantData.plant) {
      console.warn("Il campo 'plant' è mancante o invalido:",
          plantData.plant);
      return;
    }

[...]

    const existingPlantIndex = this.plants.findIndex(plant =>
        plant.plant === plantData.plant);

    if (existingPlantIndex !== -1) {
      this.plants[existingPlantIndex] = plantData;
    } else {
      this.plants.unshift(plantData);
    }

    if (this.plants.length > 0) {
      plantListContainer.classList.add("has-data");
    } else {
      plantListContainer.classList.remove("has-data");
    }

    this.updateDom();

    if (this.plants.length > 0) {
      const lastPlant =
          document.querySelector('.plant-container:last-child');
      if (lastPlant) {
        lastPlant.scrollIntoView({ behavior: 'smooth', block: 'end' });
      }
    }
  }
});
```

Given these modules are custom made, every future change can be done directly from users to better suit their usage and personal preferences, thus we omitted the **CSS** code since it is strictly relevant for the context it is used in. The only relevant advice would be that the changes on the graphics of modules should be addressed at the singular module by specifying its name before the usage of *.name* or *.container* fields, because the Smart Mirror tends to update rules accordingly to the hierarchy in use, so local changes can impact on global settings.

8 Energy and Memory Optimization

Given the resource constraints of ESP32, optimization is key:

- **Memory Management**: Data payloads sent via MQTT are kept minimal by processing sensor data locally on the ESP32 before sending updates.
- **Energy Efficiency**: ESP32 devices are put into low-power modes between sensor readings, with wake-up intervals optimized based on the frequency of readings (M seconds for soil, N seconds for plant status).

Given the resource constraints of the ESP32, careful optimization is essential to ensure smooth performance and energy efficiency in the system's operation:

- **Memory Management**: Due to the limited RAM and flash memory on the ESP32, efficient memory management plays a crucial role. Data payloads sent via MQTT are kept as compact as possible by pre-processing and filtering sensor data locally on the ESP32. Instead of sending raw data directly to the plant_brain, only the most relevant and actionable information is transmitted. This minimizes network traffic and decreases memory usage of plant_brain, allowing it to handle multiple tasks concurrently without exceeding its memory limits.
- **Energy Efficiency**: Power conservation is critical, especially for battery-powered ESP32 devices. To maximize energy efficiency, the ESP32 is programmed to enter deep sleep or light sleep modes between sensor readings. This approach significantly reduces power consumption by deactivating non-essential components when the device is idle. The wake-up intervals are carefully optimized based on the specific application needs: for example, soil moisture levels are checked every M seconds, while overall plant status (including light, temperature, and humidity) is evaluated every N seconds. By tailoring these intervals to the needs of the plant, unnecessary wake-ups are avoided, further conserving energy. Additionally, the deep sleep mode is designed to retain essential data, ensuring the system can quickly resume operation upon waking without a full reboot.

Through a combination of memory management and energy optimization strategies, the ESP32 can effectively perform its role in monitoring plant health while operating within its resource limitations. These optimizations also contribute to the long-term stability and reliability of the system, allowing for consistent performance over time despite hardware constraints.

9 Conclusions

This project successfully implements a smart plant monitoring system using ESP32 devices for sensing and data processing, with real-time data displayed on a Smart Mirror. MQTT is used for seamless and energy-efficient communication between the devices. The system allows for easy monitoring and control of plant conditions, ensuring optimal plant health with minimal manual intervention. This proves the scalability of the project to be really efficient (an ESP is easily upgradable by substituting it to a less energy-constrained, more powerful and with great costs device), the portability is one of the key point that has been considered in the development. This also lead the user to experiment with various devices and integration, given the heterogeneous ambient and the open source tools used.

10 Future Improvements

There could be a lot of improvements and changes in future builds, for example:

- **AI-Based Plant Care**: Future updates could include machine learning algorithms to predict optimal watering times and light levels based on historical data. It can also adjust, to a certain degree, the prediction to a given goal, to let the user keep an eye on the curve of production of his personal garden.
- **Water Reservoir check and notification**: This will help the user to know weather the recipient is empty or not, in order to fulfill it when necessary. It can be done by a signal given by the tank when the level of water drops after a certain fixed value.
- **Integration with a flutter app**: This will simplify user experience and give the possibility to access data even from smartphone, enhancing the portability aspect.
- **Enhanced Visualization**: By changing the visual feats, such ad modules and graphics on the display, adding more visual features to the Smart Mirror, such as graphs and historical trends of the plant's health conditions.
- **Customization**: Modify time intervals and adding plants from smart mirror, display notifications related to the water recipient and when plants are being watered, responding to the sensors with personalized messages.
- **Server Side data manipulation**: Plant Brain will be removed, and platforms like *ThingSpeak* will replace it, adding the possibility to display in smart mirror a graph related to water used and temperatures of the plants by directly taking it from the site.